# ECE 375 LAB 6

External Interrupts

**Lab Time: Friday 4-6**

*Faaiq Waqar*

*Jordan Brown*

## Introduction

Lab 6 introduces us to the subject of interrupts and requires us to implement the same bump bot algorithm that uses interrupts instead of polling. Buttons on the board will be treated like whiskers on the bot and when pressed the bot (LEDs) will behave accordingly. Lab 1 and lab 2 use the polling method to check for input but this code can be adapted to use interrupts. This lab also requires use of the LCD display to a count for the number of presses on each whisker. The bot is instructed to go forward and wait for a whisker to be triggered and then it will perform the ISR. The HitRight and HitLeft routines are still used for Lab 1 and these are the routines the program will jump to whenever an object is hit.

## Program Overview

The program starts by defining labels for several registers and assigning values to keywords used in the subroutines. Subroutines in this program are assigned to the interrupts that we hope to use with the I/O port, PORTD. In order to set up our ability to make these choices, we make a couple of decisions to manipulate data registers and memory. First, we must have the data memory locations for each of the INT0-3 locations set for the address of the subroutine we would like to call. We will also contain a reti call to return from interrupt. From here, we proceed as normal, however in the initialization routine, we must complete a couple of things. First, we must assert that our detection works on the falling edge, so we manipulate EICRA to contain corollary 10 to the bytes we want to have falling edge detection. We also needed to enable the mask for the correlated bits that we hope to use in interrupts.

From there, we use the typical initialization routine that we have seen time and time again when working with the bump bot, which includes initializing the LCD display, prepping the stack pointer, and having the program ready to switch into main. From here, the main program will infinitely loop a forward movement, which as we discussed beforehand, will only halt once an interrupt is received, and we will call the respective subroutine. Hit right and Hit left functionality derive directly from the program we used in the first lab.

### Initialization Routine

The INIT routine begins with the stack pointer being initialized because it is necessary to save the state of the processor whenever we make a jump to the ISR. The values in certain registers will be pushed onto the stack when an interrupt occurs and then popped off when we finish the ISR and return to the program. Next, we prepare DDRx for D and B by setting mpr to the proper value and sending it to the I/O space. Then we decide what type of shift in the signal will trigger the interrupt by configuring EIFRC. A value is sent to EICRA to set interrupts to trigger on the falling edge of the button signal. Finally, we turn on interrupts by setting the global interrupt flag to 1.

### Main Routine

The main routine for lab 6 is very simple, and consists of an infinite while loop that constantly drives the bot forward. At this point in the program we are just waiting for the interrupt to occur and continually jump back to the start of this routine .

### HitRight Routine

HitRight responds to the right whisker being hit, and turns away from any obstacle. When the input is received the bot will stop, reverse for a second, turn left, and then continue forward.

## HitLeft Routine

HitLeft functions in the exact same way as HitRight except it turns the bot to the right when an object is hit. With these functions we can turn the bot and control which direction it travels.

# Additional Questions

1. As this lab, Lab 1, and Lab 2 have demonstrated, there are always multiple ways to accomplish the same task when programming (this is especially true for assembly programming). As an engineer, you will need to be able to justify your design choices. You have now seen the BumpBot behavior implemented using two different programming languages (AVR assembly and C), and also using two different methods of receiving external input (polling and interrupts). Explain the benefits and costs of each of these approaches. Some important areas of interest include, but are not limited to: efficiency, speed, cost of context switching, programming time, understandability, etc.

One of the best reasons to be using the interrupt form of the bump bot program is that there is no need for the bot to be situated in a portion of the program where there is an execution of a pin check for the bumper hit. This is because there is a small but existent chance that the program will not be in a pin check execution when a user running the bump bot program , meaning that the bot might just miss the indication that a bumper is hit, and then destabilize execution. Another reason interrupts are so helpful here is that they require far less programming, which from the programmer's perspective eliminates wasteful testing and implementation, and on the side of the actual program data, we have a smaller program that fits on a smaller memory flash, which makes it better for microcontrollers with smaller amounts of memory. Interrupts however may be more complicated to work with as a newer programmer, and are limited in number per microcontroller device, so not all subroutines can be interrupt located. On the side of the polling, polling can be a great way to implement instruction sets of varying size in any respect, but lack the almost concurrent approach that we see in interrupt use, making them less viable in situations like this one in my own opinion.

2. Instead of using the Wait function that was provided in BasicBumpBot.asm, is it possible to use a timer/counter interrupt to perform the one-second delays that are a part of the BumpBot behavior, while still using external interrupts for the bumpers? Give a reasonable argument either way, and be sure to mention if interrupt priority had any effect on your answer.

To my own knowledge and understanding, no, atleast, not both at the same time, and interrupt priority has a lot to do with this. When you program an interrupt to execute, take for example INT0 which we used as our bump right interrupt subroutine, the interrupt flag in the program state is actually shut off according to AVR documentation, in order to prioritize the interrupt. Because of this, having a timer compared to a value using the clock, and sending an interrupt signal to execute some subroutine, would likely fall flat, as without the Global Interrupt Flag set, the interrupt used by the timer is rendered useless.

# Difficulties

It was difficult on a conceptual level to understand how interrupts work. We had previous knowledge from the course operating systems I, where interrupts were discussed in a shell program. From here, we were able to find out that the program was always listening for this interrupt, but then our second problem came about when we were not sure about using the PORT manipulations for the LCD display, which caused some issues, but I learned.

## Conclusion

This lab was challenging, but a lot of very important information was covered. The concepts covered are critical for success in the class and getting experience seeing how they work was helpful. Interrupts on their own are more conceptually challenging problem than an implementation challenging problem, so I appreciate that the actual programming scope of the lab was smaller to compensate for the conceptual difficulty. I wish the lab had included using a timing interrupt as well however, as it was covered in the readings and in lecture, but the scope of it is large and I think it would have helped to really implement it in lab to learn it from a more hands on input. Other than that, this was a super informative lab, and from a CS student perspective, one that had me take a step back and enjoy the thought process behind its purpose.

## Source Code

```
;***********************************************************
;*
;*	Faaiq_Waqar_and_Jordan_Brown_Lab6_sourcecode.asm
;*
;*	Work with external interupts to create TekBot Movement,
;*  According to correlated whisker hits as interrupts
;*
;*	This is the skeleton file for Lab 6 of ECE 375
;*
;***********************************************************
;*
;*	 Author: Faaiq Waqar & Jordan Brown
;*	   Date: November 15th 2019
;*
;***********************************************************

.include "m128def.inc"			; Include definition file

;***********************************************************
;*	Internal Register Definitions and Constants
;***********************************************************
```

```
.def	mpr = r16				; Multipurpose register
.def	rghtcntr = r23
.def	leftcntr = r24
.def	waitcnt = r17			; Wait Loop Counter
.def	ilcnt = r18				; Inner Loop Counter
.def	olcnt = r19				; Outer Loop Counter
.def	type = r20				; LCD data type: Command or Text
.def	q = r21					; Quotient for div10
.def	r = r22					; Remander for div10

.equ	WTime = 100				; Time to wait in wait loop

.equ	WskrR = 0				; Right Whisker Input Bit
.equ	WskrL = 1				; Left Whisker Input Bit

.equ	EngEnR = 4				; Right Engine Enable Bit
.equ	EngEnL = 7				; Left Engine Enable Bit
.equ	EngDirR = 5				; Right Engine Direction Bit
.equ	EngDirL = 6				; Left Engine Direction Bit

.equ	MovFwd = (1<<EngDirR|1<<EngDirL)	; Move Forward Command
.equ	MovBck = $00				; Move Backward Command
.equ	TurnR = (1<<EngDirL)			; Turn Right Command
.equ	TurnL = (1<<EngDirR)			; Turn Left Command
.equ	Halt = (1<<EngEnR|1<<EngEnL)		; Halt Command

;***********************************************************
;*	Start of Code Segment
```

```
;***********************************************************
.cseg                                                   ; Beginning of code segment


;***********************************************************
;*      Interrupt Vectors
;***********************************************************
.org    $0000                           ; Beginning of IVs
                rjmp    INIT                    ; Reset interrupt


                ; Set up interrupt vectors for any interrupts being used
.org    $0002
                rcall HitRight                  ; Use interrupt 0 in data space to call
                reti                                    ; Hitright
.org    $0004
                rcall HitLeft                   ; Use interrupt 1 in data space to call
                reti                                    ; HitLeft
.org    $0006
                rcall CLEARRIGHT                ; Use interrupt 2 in data space to call
                reti                                    ; ClearRight
.org    $0008
                rcall CLEARLEFT                         ; Use interrupt 3 in data space to call
                reti                                    ; Clearleft
                ; This is just an example:
;.org   $002E                           ; Analog Comparator IV
;               rcall   HandleAC                ; Call function to handle interrupt
;               reti                                    ; Return from interrupt


.org    $0046                           ; End of Interrupt Vectors


;***********************************************************
```

```
;*	Program Initialization
;***********************************************************
INIT:							; The initialization routine
		; Initialize Stack Pointer
		ldi r16, low(RAMEND)	; Prepare lower stack addr
		out SPL, r16		; Store lower stack addr
		ldi r17, high(RAMEND) ; Prepare upper stack addr
		out SPH, r17		; Store upper stack addr
		; Initialize LCD Display
		rcall	LCDInit
		rcall	CLEARRIGHT
		rcall	CLEARLEFT
		; Initialize Port B for output
		ldi		mpr, $FF		; Set Port B Data Direction Register
		out		DDRB, mpr		; for output
		ldi		mpr, $00		; Initialize Port D Data Register
		out		PORTD, mpr		; so all Port D inputs are Tri-State
		; Initialize Port D for input
		ldi		mpr, $00		; Set Port D Data Direction Register
		out		DDRD, mpr		; for input
		ldi		mpr, $FF		; Initialize Port D Data Register
		out		PORTD, mpr		; so all Port D inputs are Tri-State
		; Initialize external interrupts
			; Set the Interrupt Sense Control to falling edge
		ldi		mpr,
(1<<ISC01)|(0<<ISC00)|(1<<ISC11)|(0<<ISC10)|(1<<ISC21)|(0<<ISC20)|(1<<ISC31)|(0<<ISC30)
		sts		EICRA, mpr
		; Configure the External Interrupt Mask
		ldi		mpr, (1<<INT0)|(1<<INT1)|(1<<INT2)|(1<<INT3)
		out		EIMSK, mpr
```

```
		; Turn on interrupts
			; NOTE: This must be the last thing to do in the INIT function
			sei
;***********************************************************
;*	Main Program
;***********************************************************
MAIN:							; The Main program

		; TODO: ???
		ldi		mpr, MovFwd		; Move the robot forward infiniely
		out		PORTB, mpr		; Output to the display port
		rjmp	MAIN			; Create an infinite while loop to signify the
								; end of the program.

;***********************************************************
;*	Functions and Subroutines
;***********************************************************

;-----------------------------------------------------------
;	You will probably want several functions, one to handle the
;	left whisker interrupt, one to handle the right whisker
;	interrupt, and maybe a wait function
;------------------------------------------------------------

;-----------------------------------------------------------
; Func: Template function header
; Desc: Cut and paste this and fill in the info at the
;		beginning of your functions
;-----------------------------------------------------------
HITRIGHT:							; Begin a function with a label
```

```
push   mpr                  ; Save mpr register
push   waitcnt              ; Save wait register
in            mpr, SREG     ; Save program state
push   mpr                  ;

; Move Backwards for a second
inc           rghtcntr
rcall  WRITERIGHT
ldi           mpr, MovBck   ; Load Move Backward command
out           PORTB, mpr    ; Send command to port
ldi           waitcnt, WTime ; Wait for 1 second
rcall  WAITSEC              ; Call wait function

; Turn left for a second
ldi           mpr, TurnL    ; Load Turn Left Command
out           PORTB, mpr    ; Send command to port
ldi           waitcnt, WTime ; Wait for 1 second
rcall  WAITSEC              ; Call wait function

; Move Forward again
ldi           mpr, MovFwd   ; Load Move Forward command
out           PORTB, mpr    ; Send command to port

ldi           mpr, $FF
out           EIFR, mpr

pop           mpr           ; Restore program state
out           SREG, mpr     ;
pop           waitcnt       ; Restore wait register
pop           mpr           ; Restore mpr
```

```
        ret                         ; Return from subroutine
        ; Save variable by pushing them to the stack

HITLEFT:                                          ; Begin a function with a label
        push    mpr                 ; Save mpr register
        push    waitcnt             ; Save wait register
        in      mpr, SREG           ; Save program state
        push    mpr                 ;

        ; Move Backwards for a second
        inc     leftcntr
        rcall   WRITELEFT
        ldi     mpr, MovBck         ; Load Move Backward command
        out     PORTB, mpr          ; Send command to port
        ldi     waitcnt, WTime      ; Wait for 1 second
        rcall   WAITSEC             ; Call wait function

        ; Turn right for a second
        ldi     mpr, TurnR          ; Load Turn Left Command
        out     PORTB, mpr          ; Send command to port
        ldi     waitcnt, WTime      ; Wait for 1 second
        rcall   WAITSEC             ; Call wait function

        ; Move Forward again
        ldi     mpr, MovFwd         ; Load Move Forward command
        out     PORTB, mpr          ; Send command to port

        ldi     mpr, $FF
        out     EIFR, mpr
```

```
            pop         mpr         ; Restore program state
            out         SREG, mpr   ;
            pop         waitcnt     ; Restore wait register
            pop         mpr         ; Restore mpr
            ret                     ; Return from subroutine

            ; Save variable by pushing them to the stack

            ; Execute the function here

            ; Restore variable by popping them from the stack in reverse order

WAITSEC:
            push    waitcnt         ; Save wait register
            push    ilcnt           ; Save ilcnt register
            push    olcnt           ; Save olcnt register

Loop:   ldi         olcnt, 224      ; load olcnt register
OLoop:  ldi         ilcnt, 237      ; load ilcnt register
ILoop:  dec         ilcnt           ; decrement ilcnt
            brne    ILoop           ; Continue Inner Loop
            dec         olcnt       ; decrement olcnt
            brne    OLoop           ; Continue Outer Loop
            dec         waitcnt     ; Decrement wait
            brne    Loop            ; Continue Wait loop

            pop         olcnt       ; Restore olcnt register
            pop         ilcnt       ; Restore ilcnt register
            pop         waitcnt     ; Restore wait register
            ret                     ; Return from subroutine
```

```
WRITERIGHT:

		push	mpr					; Save MPR state


		ldi		YL, low($0100) ; Load LCD Read Location

		ldi		YH, high($0100)		; In data mem to Y

		ld		mpr, Y				; Load value form Y into MPR

		inc		mpr						; Increment MPR

		st		Y, mpr				; Store MPR in data mem at Y

		rcall	LCDWrLn1			; Write to screen


		pop		mpr						; Restore the value in MPR

		ret


WRITELEFT:

		push	mpr					; Save the multi purpose reg


		ldi		YL, low($0110) ; Load address 0110 to Y

		ldi		YH, high($0110)

		ld		mpr, Y				; Save the data from the data loc

		inc		mpr						; Increment by One

		st		Y, mpr				; Store in data mem

		rcall	LCDWrLn2			; Display


		pop		mpr						; Restore Value in MPR

		ret


CLEARRIGHT:


		push	mpr
```

```
		push	waitcnt			; Save wait register
		in		mpr, SREG		; Save program state
		push	mpr				; Save program state to stak

		ldi		YL, low($0100) ; Load data mem location for lcdr
		ldi		YH, high($0100)
		ldi		mpr, $30		; Load MPR with ASCII 0
		st		Y, mpr			; Store into data memory
		rcall	LCDWrLn1		; Display

		ldi		mpr, $FF		; Terminate Queued Interrupts
		out		EIFR, mpr		; Output onto the flag reg for int

		pop		mpr		; Restore program state
		out		SREG, mpr	;
		pop		waitcnt		; Restore wait register
		pop		mpr
		ret

CLEARLEFT:
		push	mpr
		push	waitcnt			; Save wait register
		in		mpr, SREG		; Save program state
		push	mpr				; Save program state to the stack

		ldi		YL, low($0110) ; Pair location in data mem to Y
		ldi		YH, high($0110)
		ldi		mpr, $30		; Load with ASCII value 0
		st		Y, mpr			; store intodata location
		rcall	LCDWrLn2		; Display
```

```
        ldi     mpr, $FF        ; Eliminate Queues
        out     EIFR, mpr

        pop     mpr         ; Restore program state
        out     SREG, mpr   ;
        pop     waitcnt     ; Restore wait register
        pop     mpr
        ret
;***********************************************************
;*      Stored Program Data
;***********************************************************

; Enter any stored data you might need here

;***********************************************************
;*      Additional Program Includes
;***********************************************************
.include "LCDDriver.asm"
```